Optimizing the Performance of the FM-Index for Large-Scale Data

Eddie Ferro
Department of Computer and Information Science and Engineering, Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Christina Boucher[1]
Department of Computer and Information Science and Engineering, Herbert Wertheim College of Engineering, University of Florida, Gainesville, FL, USA

Abstract

The FM-index is a fundamental data structure used in bioinformatics to efficiently search for strings and index genomes. However, the FM-index can pose computational challenges, particularly in the context of large-scale genomic datasets, due to the complexity of its underlying components and data encodings. In this paper, we present a comprehensive review of efficient variants of the FM-index and the encoding strategies used to improve performance. We examine hardware-accelerated techniques, such as memory-efficient data layouts and cache-aware structures, as well as software-level innovations, including algorithmic refinements and compact representations. The reviewed work demonstrates substantial gains in both speed and scalability, making methods that use the FM-index more practical for high-throughput genomic applications. By analyzing the trade-offs and design choices of these variants, we highlight how combining hardware-aware and software-centric strategies enables more efficient FM-index construction and usage across a range of bioinformatics tasks.

2012 ACM Subject Classification Theory of computation → Data compression

Keywords and phrases FM-Index Acceleration, Run-Length Encoding, Suffix Array Optimization, Burrows-Wheeler Transform, Efficient Backward Search

Digital Object Identifier 10.4230/OASIcs.Manzini.2025.6

Acknowledgements I want to thank Travis Gagie for helpful suggestions.

1 Introduction

The introduction of the FM-index in 2000 by Ferragina and Manzini [14] was a novel concept that sought to combine the compression of word-based indices with the efficiency in searching for full-text indices. This revolutionary data structure, built on the Burrows-Wheeler Transform (BWT), offered a compact representation of text while maintaining efficient support for pattern matching queries. Using the properties of the BWT and the Suffix Array (SA), the FM-index enabled search operations, such as locating and counting occurrences of a pattern, to be performed in linear time with respect to the pattern length in a technique called backward search. Furthermore, the FM-index achieved space savings by compressing text data proportional to the empirical entropy of the input, making it an ideal tool for applications in data compression, bioinformatics, and information retrieval.

Despite its advantages, the traditional FM-index encountered significant performance bottlenecks when applied to large-scale highly repetitive datasets. These datasets, common in fields like bioinformatics and genomics, often feature long stretches of repeated or similar sequences that exacerbate the trade-offs between time efficiency and memory usage. The primary challenge lay in balancing the need for compact data representation with the ability

[1] Corresponding author

to perform search operations efficiently. The original FM-index construction [14], while theoretically robust, struggled with memory overhead when applied to repetitive data due to its reliance on suffix arrays and wavelet trees, which scale poorly in such scenarios. Additionally, the cache-unfriendly random-memory accesses required to the BWT during backward search steps further limited its practical efficiency on modern hardware.

These limitations spurred a wave of research focused on enhancing the FM-index by introducing computational techniques that leverage the repetitive nature of input data to optimize storage and query performance. For example, the Run-Length FM-index (RLFM) introduced by Mäkinen and Navarro [26, 22] applied run-length encoding to the Burrows-Wheeler Transform (RLBWT), significantly reducing space requirements in repetitive datasets while maintaining efficient backward searches. Similarly, Grammar Compression Induced by Suffix Sorting (FIGISS) [10] employed grammar-based compression to accelerate backward search operations, particularly for long patterns, by replacing substrings with nonterminal symbols derived from suffix sorting. The Prefix-Free Parsing FM-index (PFP-FM) [17], in contrast, utilized a parsing technique to divide text into phrases, allowing search operations to process larger segments of the pattern at each step, thereby reducing the number of backward search iterations needed.

These advancements collectively aim to address the inherent space-time trade-offs in FM-index operations, offering tailored solutions for various use cases and dataset types. Each method innovatively adapts the FM-index framework to better suit the demands of large-scale repetitive data, ensuring its continued relevance and efficiency in modern applications.

This review provides an in-depth exploration of these advancements, focusing on the theoretical innovations and practical implementations that have significantly improved the FM-index's performance. By analyzing the methods used to compress repetitive data, accelerate backward search operations, and enhance scalability, this paper aims to present a comprehensive overview of the current state of FM-index technology and its applications in handling large and repetitive datasets.

2 Preliminaries

Throughout this section, we will discuss key data structures and algorithms used for string processing, focusing on their foundational definitions and properties. These concepts are vital for understanding the time and space efficiency of various text indexing methods, especially in the context of large datasets. Starting with basic string definitions, we will build up to more advanced structures such as suffix trees, arrays, and compression techniques, highlighting their roles in efficient pattern matching and data compression.

2.1 Basic Definitions

A string T is a finite sequence of symbols $T = T[1..n] = T[1] \cdots T[n]$ over an alphabet $\Sigma = \{c_1, \ldots, c_\sigma\}$ whose symbols can be unambiguously ordered. We denote ε as the empty string, $|T|$ as the length of T, σ as the size of Σ, and the rotation of T is a transformation where the characters are circularly shifted so that the first character becomes the last character of the rotated string. A bitvector B is defined as a finite sequence $B = B[1..n] = B[1] \cdots B[n]$ over the alphabet $\Sigma = \{0, 1\}$. Unlike standard arrays, which typically require $\Theta(n \log \sigma)$ bits to store n elements on an alphabet of size σ, a bitvector uses only n bits, making it a highly space-efficient representation when working over binary data.

We define the substring of $T[i] \cdots T[j]$ of T as $T[i..j]$ starting at position i and ending in position j, with $T[i..j] = \varepsilon$ if $i > j$. For a string T and $1 \leq i \leq n$, $T[1..i]$ is called the i-th prefix of T, and $T[i..n]$ is called the i-th suffix of T. We call a prefix $T[1..i]$ of T a *proper prefix* if $1 \leq i < n$. Similarly, we call a suffix $T[i..n]$ of T a *proper suffix* if $1 < i \leq n$. Given a set of strings $\mathcal{S}$, $\mathcal{S}$ is *prefix-free* if no string in $\mathcal{S}$ is a prefix of another string in $\mathcal{S}$.

We use $\prec$ to denote the lexicographic ordering of two strings, which is defined as follows. For two strings $T_2[1..m]$ and $T_1[1..n]$, $T_2 \prec T_1$ if T_2 is a proper prefix of T_1, or there exists an index $1 \leq i \leq min(n, m)$ such that $T_2[1..i-1] = T_1[1..i-1]$ and $T_2[i] < T_1[i]$. We define the co-lexicographic order of two strings, T1 and T2, as the lexicographic order obtained by reading the strings from right to left. We assume that the string T is terminated by a special symbol \$ that is lexicographically smaller than any other symbol in Σ, unless otherwise specified. This provides the technical convenience of allowing all rotations of a string and all suffixes of the same string to have coinciding lexicographic ordering.

We define a `count` query and a `locate` query for some pattern P as follows. $\texttt{count}(P)$ returns the number of occurrences of the pattern in T. $\texttt{locate}(P)$ returns all positions in T where this pattern occurs. These queries can be answered without access to the text by using an index of the text.

Throughout this survey, the time and space complexities will be specified in the RAM model of computation, where programs are run on a random-access memory where words of $w = \log n$ bits are accessed and manipulated in constant time. Therefore, we will generally measure space in words as well. Moreover, in this model, we assume that all ordinary arithmetic and logical operations are performed in constant time.

2.2 Rank and Select

For a string T of length n, a symbol $c \in \Sigma$, and a position i in T, the rank function $T.\mathsf{rank}_c(i)$, is defined as the number of occurrences of c in $T[1..i]$. Given an integer i, the select function $T.\mathsf{select}_c(i)$, returns the position of the i-th occurrence of c in T, if it exists.

For a bitvector B, we will instead define the rank function $B.\mathsf{rank}(i)$, as the number of occurrences of 1 in $B[1..i]$, and the select function, $B.\mathsf{select}(i)$, as the position in B of the i-th occurrence of 1. We note that for a bitvector we automatically assume that the character of interest is 1.

2.3 The Suffix Trie and Suffix Array

The *suffix tree* [39, 24, 1] is a compressed trie storing the suffixes of a string T, where paths with a single child are compressed into a single node with the edge labeled by the concatenation of the compressed edge symbols. As there are n suffixes and no suffix can be repeated twice, there are exactly n leaves. Furthermore, since there are no 1-degree nodes and n leaves, the suffix tree can be stored in the $\mathsf{O}(n)$ space. Assuming that T is defined by a constant-size alphabet or an alphabet consisting of integers in a polynomial range, it can also be built in $\mathsf{O}(n)$ time [39, 24, 38, 12].

The *suffix array* [23] (denoted as SA) was introduced to reduce the space consumption of the suffix tree. For a given string T of length n, the SA is the array $\mathsf{SA} = [1..n]$ defined such that $\mathsf{SA}[i]$ is the text position of the i-th suffix of T in lexicographical order. The theoretical space required by the SA is still $\mathsf{O}(n)$, but only a single integer is stored for each suffix, so the practical space is much smaller than that of the suffix tree. Another benefit of SA is that it is possible to build without having to build the suffix tree in $\mathsf{O}(n)$ time [19].

Both the suffix tree and the suffix array can be used to match a pattern $P = [1..m]$ to all occurrences in T. In the suffix tree, the process involves traversing the nodes from the root symbols matching P to T, eventually ending at some node or on the edge between two nodes. All paths that follow from this point represent a suffix that starts with the pattern P and as such the pattern has been matched. This is done in $\mathsf{O}(m)$ time assuming that the child node that extends the path can be located in constant time through the use of hashing or some $\mathsf{O}(1)$ lookup data structure. Given the suffix array of T, the query pattern can be found by performing a binary search over the suffix array. The binary search takes $\mathsf{O}(m \log n)$ time and returns an interval in the suffix array that represents all occurrences of the query pattern in T. A binary search can be used because the suffixes are sorted lexicographically, so suffixes that start with the same prefix are grouped together. This bound can be reduced to $\mathsf{O}(m + \log n)$ with the use of additional data structures [23].

We now define the inverse of SA as *inverse suffix array* (ISA). Given a suffix array of T, SA, the corresponding inverse suffix array, ISA, is defined as $\mathsf{ISA}[i] = j$ if and only if $\mathsf{SA}[j] = i$. In other words, this structure returns the position of a suffix in the SA given its starting position in the text. Furthermore, many applications, such as FM-index, that make use of SA do not store the entire SA and instead store a sample of SA values, which is called the `sampled SA`.

2.4 Burrows-Wheeler Transform and LF-mapping

The Burrows-Wheeler Transform ($\mathsf{BWT}[1..n]$) [8] of a string $T[1..n]$ is a reversible permutation of the characters of the string. This means that it is possible to retrieve the original string, T, from its BWT. Let us denote the matrix of all sorted rotations of T as $\mathrm{M}_{\mathsf{BWT}}$, and F and L as the *First* and *Last* column of this matrix, respectively. By definition, L is the BWT of T and F contains all the characters in T in lexicographic order. If we were to only consider for each row of $\mathrm{M}_{\mathsf{BWT}}$ from the first character of each rotation to the special character \$ (the last character in T), we would have the suffixes of the original input in lexicographic order. That is exactly what is stored by the SA and this insight allows for the BWT to be constructed from the SA. More formally, this is defined as $\mathsf{BWT}[i] = T[n] = \$$ if $\mathsf{SA}[i] = 1$ and $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$ otherwise. More precisely, the BWT is storing the characters that occur before each suffix in lexicographic order. The BWT can be built directly in $\mathsf{O}(n)$ time and $\mathsf{O}(n \log \sigma)$ bits of space [25], or built from the SA of T using the formulas above.

A key property of the BWT is that the i-th occurrence of a character in L and the i-th occurrence of the same character in F are the same character in T. This property comes from the fact that the BWT is constructed by lexicographically ordered cyclic rotations. If we consider the row of the first occurrence of a character c in L, the character, say c', in F in that same row is the character that follows c in the original input. Since this is the first occurrence of c, this character, c', is also the lexicographically smallest character that follows any occurrence of c. During the sorting of F all c occurrences are grouped together and their order is determined by what follows them and since c' is the lexicographically smallest, the first occurrence of c in L is the first occurrence of c in F. Next, we denote $\mathsf{C}[c]$ as the number of suffixes starting with a character smaller than c, which can be derived from F by counting the number of occurrences of each character. We now define the LF-mapping of the BWT as follows:

▶ **Definition 1** ([14]). *Given the BWT of a string T and the array C, the LF-mapping function is* $\mathsf{LF}(i) = \mathsf{C}[c] + \mathsf{BWT.rank}_c(i)$, *where* $c = \mathsf{BWT}[i]$.

The LF-mapping function relates the row of the i-th occurrence of a character in L to the row of the i-th occurrence of the same character in F and makes it possible to reconstruct the string T from its BWT. To reconstruct T, we set an iterator $t = 1$ and $T[n] = \mathsf{BWT}[1]$ and for each $i = n-1, \ldots, 1$ do $t = \mathsf{LF}(t)$ and $T[i] = \mathsf{BWT}[t]$. The way this works is that the LF-mapping is used to find the row of $\mathrm{M}_{\mathsf{BWT}}$ where F contains the same character as $\mathsf{L}[t] = \mathsf{BWT}[t]$ and since BWT stores the character that occurs before the character in F in T, then BWT in this row gives us the next character in reconstruction. The LF-mapping is essential in the FM-index and all related indexes, as will be shown.

2.5 FM-Index and Backwards Search

The FM-index by Ferragina and Manzini [14] is a compressed full-text index and the first self-index, an index that also replaces the text entirely. Formally, it makes use of the BWT and the SA to efficiently support `count` and `locate` queries, essential queries in pattern matching. The BWT of the input string supports `count` queries through the use of LF-mapping in a technique called backward search [14]. Backward search is formally defined as follows.

▶ **Definition 2** ([14]). *Given a pattern $P = [1..p]$, the backward search consists of p steps that follow the same procedure: at the i-th step, sp stores the position of the first row of $\mathrm{M}_{\mathsf{BWT}}$ prefixed by $P[i, p]$ while ep stores the position of the last row of M_{BWT} prefixed by $P[i, p]$. To advance from i to $i-1$, we use LF-mapping on sp and ep as follows $sp = \mathsf{C}[c] + \mathsf{BWT.rank}_c(sp-1) + 1$ and $ep = \mathsf{C}[c] + \mathsf{BWT.rank}_c(ep)$. At the final step, if $ep < sp$, then the pattern does not occur in the input text; Otherwise, they represent the range of rows of M_{BWT} prefixed by P.*

Using the result of the backward search, a `count` query can be answered simply by providing the size of the range, or $ep - sp + 1$. Note that none of the steps of the backward search rely on the complete BWT matrix $\mathrm{M}_{\mathsf{BWT}}$, it is sufficient to retain only the BWT, which is commonly stored in compressed form, and the array $\mathsf{C}[1..\sigma]$. It is also easy to see that the *backward search* algorithm takes $\mathsf{O}(|P|)$ time, assuming that each $LF(i)$ computation can be performed in constant time. Consequently, a `count` query can be answered by the FM-index in $\mathsf{O}(|P|)$ time under this assumption [14, 13]. The SA of the input string enables all occurrences to be efficiently located. A range in BWT directly corresponds to a range in SA, which means that locating the occurrences would be trivial with the full SA. However, in FM-index only a `sampled` SA is stored. We formally define the process of locating occurrences with a `sampled` SA as follows.

▶ **Definition 3** ([14]). *Given an interval $[s_p, e_p]$, locating occurrences consists of $occ = e_p - s_p + 1$ steps that all have the following invariant: at the i-th step, where $x = s_p + i$, if $\mathsf{SA}[x]$ exists then return that value. Otherwise perform $x = \mathsf{LF}(x)$ a total of v times until $\mathsf{SA}[x]$ exists and then return $\mathsf{SA}[x] + v$.*

The FM-index stores SA values at regular positions in the text T. Specifically, for some sampling parameter, s, each position $\mathsf{SA}[i]$ is kept if $\mathsf{SA}[i] \bmod s \equiv 0$. As a result, locating each of the occ occurrences requires traversing up to s LF-mapping steps, yielding a total time complexity of $\mathsf{O}(occ \cdot s)$. In the original implementation, $s = \log^{1+\epsilon}(n)$ for some $\epsilon > 0$. The FM-index occupies $\frac{5nH_K}{\log n} + \mathsf{O}(\frac{n \log \sigma}{\log n})$ words of space.

2.6 Wavelet Trees

The *wavelet tree* is a data structure by Grossi et al. [16] that can be used to store a string T over an alphabet Σ of size σ achieving $\frac{n \log \sigma}{\log n} + \mathsf{o}(\frac{n \log \sigma}{\log n})$ words of space and $\mathsf{O}(\log \sigma)$ time rank and select operations over the input text. To achieve these bounds, we assume

that the underlying bitvectors of the wavelet tree support constant-time rank and select. A wavelet tree is made up of $\log \sigma$ bitvectors organized in a balanced binary tree structure. The bitvector at the root, B_{root}, is the same size as the input T and is defined by choosing a character $v \in \Sigma$, usually the median character in lexicographic order, such that $B_{root}[i] = 0$ if $T[i] < v$ and $B_{root}[i] = 1$ otherwise. The positions marked as 0 form the subsequence that is used for the left child and the positions marked as 1 form the subsequence for the right child. The bitvectors for each of these children are created in the same way as the root, keeping in mind that each child's subsequence is on half the alphabet of the parent sequence. This partitioning is performed recursively until the bitvector of a node consists entirely of 0s or 1s, which denotes that this node is a leaf. Answering the query $T.\mathsf{rank}_c(i)$ is done by traversing the tree and computing rank on the bitvector at each level. We initialize $pos_0 = i$ and continue down the left subtree with $pos_{j+1} = B.\mathsf{rank}_0(pos_j)$ if $c \prec v$ or down the right subtree with $pos_{j+1} = B.\mathsf{rank}_1(pos_j)$ otherwise. The query stops when a leaf is reached and the result is pos_h, where h is the height of the tree. The query $T.\mathsf{select}_c(i)$ is answered by first identifying the leaf node for c, which can be done using a rank query. Then, the tree is traversed from the leaf node to the root through select queries on every node's bitvector. We initialize $pos_h = i$ and if the current node is the left child of its parent, we traverse with $pos_j = B.\mathsf{select}_0(pos_{j+1})$; otherwise, we traverse with $pos_j = B.\mathsf{select}_1(pos_{j+1})$. The result of the query is then pos_0, which we get once we reach the root.

2.7 Prefix-Free Parsing

Prefix-Free Parsing (PFP) is a technique created by Boucher et al. [5] that compresses and replaces the input before constructing the index. This allows large data sets, such as pangenomes, to be indexed without storing the full input, reducing practical memory requirements. PFP takes as input a string T of length n, and two parameters, which we denote w and p. It produces a parse and a dictionary of T, denoted as P and D, respectively, which can be used to reconstruct the original input. The dictionary consists of unique overlapping phrases stored in their lexicographic ordering, and the parse gives the order in which these phrases appear in the input. The Prefix-Free in the name of the algorithm is because the output has the property that none of the suffixes of length greater than w of the phrases in D is a prefix of any other. This concept is formalized in the following lemma. We refer to PFP of T as PFP(T), consisting of the dictionary D and the parse P.

▶ **Lemma 4** ([5]). *If we have a string T and PFP(T), then the set $\mathcal{S}$ of distinct proper phrase suffixes of length at least w of the phrases in D is a prefix-free set.*

In order to construct these data structures, PFP first appends w copies of \$ – a special character that is lexicographically smaller than any element in Σ – to both the beginning and end of T. For the sake of explanation, we consider the string $T' = \$^w T \$^w$. Next, we define the set of trigger strings E, which determine how to parse T'. Given a parameter p, we construct the set of trigger strings by computing the Karp-Rabin hash, $H_p(t)$, of substrings of length w by sliding a window over T'. E is then the set of substrings $t = T'[i..i + w - 1]$, where $H_p(t) \equiv 0$ or $t = \$^w$. Then, using E, we can scan through T', store every substring that starts and ends with a trigger string in the dictionary, and then only keep the unique phrases. The parse is then a string of integers, where each integer is a position in the dictionary, and the string indicates the order in which these phrases appear in T'. Note that by definition of the trigger strings and by appending w copies of \$, that T' starts and ends with a trigger string. PFP can be used as a preprocessing step to build data structures such as the BWT, SA, and LCP.

3 Optimization via Sampling and Run-Length Encoding

3.1 Run-Length Encoding the FM-Index

Mäkinen and Navarro introduced a major improvement in the space requirements of the FM-index in 2004 [26, 22] with the RLFM index. The key idea behind the RLFM index is to exploit the structure of the run-length encoded Burrows–Wheeler Transform (RLBWT). The RLBWT is a data structure that encodes the BWT in $\mathsf{O}(r)$ words, where r is the number of runs in the BWT, and supports $LF(i)$ queries in $\mathsf{O}(\log\log_w(\sigma + n/r))$ time. Here, a run in a string is defined as a maximal sequence of consecutive identical characters; thus, the number of runs corresponds to the number of such maximal contiguous blocks in the BWT. For example, if BWT = $\$aaacgggttacc$, then this BWT has seven runs. The construction of the BWT matrix groups similar prefixes together, indicating shared context in the input. This common context leads to the repeating of the same letter for several rows in BWT and is especially true in highly repetitive data where the number of runs, r, can be significantly less than the number of characters in the BWT. The RLBWT can be constructed indirectly or directly. The indirect construction of the RLBWT, which is the type of construction done in the RLFM index, is done by first constructing the BWT and then run-length encoding it. This can be done $\mathsf{O}(n)$ time and $\mathsf{O}(|\mathsf{PFP}|)$ space [5] or $\mathsf{O}(n)$ time and $\mathsf{O}(\frac{n\log\sigma}{\log n})$ space[3, 25]. Direct construction of the RLBWT can be done in optimal time ($\mathsf{O}(n + r\log r)$ time) and BWT-runs bounded space ($\mathsf{O}(r)$ space)[29].

Taking advantage of the compressibility of BWT through run-length encoding, the RLFM index achieves substantial space savings while maintaining efficient `count` query performance. Mäkinen and Navarro run-length encode the BWT by replacing it with a bitvector, B, of length the size of the input that contains a 1 at the start of every run in the input, and a string, S, which lists all the characters that make up each run in order. Consider the earlier example where BWT = `$aaacgggttacc`. This sequence is replaced by two components: a bitvector B =`1100110010110`, where each `1` denotes the start of a new run, and a string S = `$acgtac`, which stores the distinct characters corresponding to each run in the order of appearance in BWT. The authors demonstrate that the LF-mapping can still be computed using only these representations. To support this, a new bitvector B' is derived by performing a stable sort of the runs encoded in B, where the sorting is based on the lexicographic order of the associated character of each run in S, and stability ensures that the original relative order among runs of the same character is preserved. For the given example, this results in B' = `1100111010010`. Additionally, instead of storing S directly, a bitvector, S_c, is stored for each character c storing a `1` at every index where c occurs in S. In our example, we would then have the following bitmaps: $S_{\$}$ = `1000000`, $S_{\mathtt{a}}$ = `0100010`, $S_{\mathtt{c}}$ = `0010001`, $S_{\mathtt{g}}$ = `0001000`, and $S_{\mathtt{t}}$ = `0000100`. Mäkinen and Navarro compute the LF-mapping in the RLFM index by presenting an equivalent equation to $\mathsf{C}[c] + \mathsf{BWT.rank}_c(i)$ that uses B, B', and the S_c bitmaps. We denote that C_S is the same as the array C, but over the string S rather than the input. The authors present the following lemma.

▶ **Lemma 5** ([22]). *Given S and B' for some BWT for any $c \in \Sigma$:*

$$\mathsf{C}[c] = B'.\mathsf{select}(\mathsf{C}_S[c] + 1) - 1$$

To prove this lemma, the authors establish that since $\mathsf{C}_S[c]$ gives the number of runs of characters smaller than c and B' contains the runs sorted lexicographically, then $B'.\mathsf{select}(\mathsf{C}_S[c]+1)$ is the position in B' corresponding to the first run of c. It follows that $B'.\mathsf{select}(\mathsf{C}_S[c]+1)-1$ is the sum of the run lengths of all characters smaller than c, which is equivalent to the number of characters smaller than c in BWT, or $\mathsf{C}[c]$. The authors used this to then give the equation for the LF-mapping in RLBWT as follows.

▶ **Lemma 6** ([22]). *Given S, B', and B of some RLBWT. For any $c \in \Sigma$ and position $1 \leq i \leq n$ that is the last position of a run:*

$$\mathsf{C}[c] + \mathsf{BWT}.\mathsf{rank}_c(i) = B'.\mathsf{select}(\mathsf{C}_S[c] + 1 + S_c.\mathsf{rank}(B.\mathsf{rank}(i))) - 1$$

The proof starts with the fact that $B.\mathsf{rank}(i)$ gives the position of the run that finishes at i in S, which means that $S_c.\mathsf{rank}(B.\mathsf{rank}(i))$ is the number of runs of c in the BWT from 1 to i. Therefore, $C_S[c]+1+S_c.\mathsf{rank}(B.\mathsf{rank}(i))-1$ points to a position in B' within the runs of c or to the character immediately after, which means that $B'.\mathsf{select}(C_S[c]+1+S_c.\mathsf{rank}(B.\mathsf{rank}(i)))-1$ is the starting position for the runs of c plus the sum of the run lengths of c in $\mathsf{BWT}[1..i]$. This is the same as $C[c]+S_c.\mathsf{rank}(B.\mathsf{rank}(i))$. If $\mathsf{BWT}[i] \neq c$, then this is the LF-mapping function. Otherwise, it is $B'.\mathsf{select}(C_S[c] + 1 + S_c.\mathsf{rank}(B.\mathsf{rank}(i))) + i - B.\mathsf{select}(B.\mathsf{rank}(i))$, because the position we are looking for is within the run c, so the added term of $i - B.\mathsf{select}(B.\mathsf{rank}(i))$ gives how far into the run i is.

Mäkinen and Navarro presented an improved FM-index that when performing solely `count` queries required less space than the FM-index and achieved the same query time. The authors tested their implementation against the implementation of the FM-index, the original SA, the Compressed Suffix Array (CSA) [36], and the LZ self-index [27] using random patterns from the index text. Their index required 0.63 of the original input size, which is less than every competing method except for CSA with tuned parameters. In terms of `count` query time, RLFM was only slower than the implementation of the FM-index, requiring around 0.1 msec to `count` occurrences for the longest pattern length compared to around 0.05 msec. In order to answer `locate` queries, RLFM uses the same SA sampling techniques as the FM-index, which requires more than $\mathsf{O}(r)$ words of space. The RLFM with `locate` therefore requires more space than CSA and less than the FM-index, but had comparable `locate` query times across all pattern lengths.

3.2 Sampling the Suffix Array: the r-index

Although `count` queries could be performed in $\mathsf{O}(r)$ space with the RLFM index, it was not until the development of the r-index index by Gagie et al. [15] that `locate` queries could also be supported within $\mathsf{O}(r)$ words of space. This is because the SA values are needed to answer a `locate` query, and storing the full SA requires $\mathsf{O}(n)$ space. Even when sampling the SA values at a constant rate, as was done in the original FM-index, the space required would be linearly proportional to the input size. Policriti and Prezza's Toehold Lemma [35, 34] enabled SA values to be sampled once per run while still supporting `locate` queries. The lemma is as follows.

▶ **Lemma 7** ([15]). *With $\mathsf{O}(r)$-words and given a pattern $P[1..m]$, the interval $\mathsf{BWT}[s_p..e_p]$ of all occurrences of P in the input T and the value $\mathsf{SA}[p]$ of some position p in the interval $[s_p, e_p]$ can be found in $\mathsf{O}(\log\log(n/r))$ time.*

The Toehold Lemma states that by storing a predecessor data structure in $\mathsf{O}(r)$ words containing SA values at the start and end of each run in the BWT, it is possible to maintain a "toehold" – that is, an SA value for at least one position within the current BWT interval – at every step of the backward search. To prove this, assume that a pattern P, so far matched with $i+1..m$, the interval $\mathsf{BWT}[s_p, e_p]$ has been found, and $\mathsf{SA}p$ is known for some index p in the BWT interval. If the next character that matches $P[i]$ is the same as $\mathsf{BWT}[p]$, then the value of the SA for some index p' in the new interval $[s'_p, e'_p]$ can be found by setting $p' = \mathsf{LF}(p)$ and $\mathsf{SA}[p'] = \mathsf{SA}[p] - 1$. If instead $\mathsf{BWT}[p] \neq P[i]$ and P occurs in the text,

then there must exist an index u such that $s_p \leq u \leq e_p$ and $\mathsf{BWT}[u] = P[i]$, as well as at least one index in $[s_p, e_p]$ for which BWT differs from $P[i]$. This implies that the next interval intersects a run boundary and, therefore, a sampled SA value is stored for this position. Finding this SA value requires a predecessor query to the earlier defined predecessor structure that can be completed in $\mathsf{O}(\log\log(n/r))$ time. Therefore, the lemma is proven using induction. Improvements to the Toehold lemma by Bannai et al. [2] show that the SA only needs to be sampled at run starts for the proof to remain valid, which helps improve practical performance but the $\mathsf{O}(r)$ words of space bound remains.

In their paper, Gagie et al. show that knowing a position in the BWT and its SA value (through Lemma 7), it is possible to compute the SA values of neighboring rows in $\mathsf{O}(\log\log(n/r))$ time. To do this, the authors used the definition of the ϕ-array of Kärkkäinen et al. [18].

▶ **Definition 8.** *Given a string S, SA of S, and ISA of S, we define the function ϕ as follows, $\phi(i) = \mathsf{SA}[\mathsf{ISA}[i] - 1]$ if $\mathsf{ISA}[i] \neq 1$, and $\phi(i) = \mathsf{SA}[n]$ otherwise. This allows us to move to the SA value before the current one via ϕ. We define ϕ^{-1} symmetrically, allowing us to move to the next SA value from the current one.*

The presented lemma states that by storing an $\mathsf{O}(r)$ space predecessor data structure it is possible to evaluate ϕ and ϕ^{-1} in $\mathsf{O}(\log\log(n/r))$ time. The predecessor data structure stores for every position i in the text corresponding to a start of the run the positions in the text (the SA values) of the preceding and following position of the position i's corresponding position in BWT, q. Essentially, the predecessor data structure, N, can be defined as follows for every i: $N[i] =< \mathsf{SA}[q-1], \mathsf{SA}[q+1] >$ for $i = \mathsf{SA}[q] - 1$. To then find the SA values neighboring for a given position, k, in the text (equivalently position p in the BWT), first a predecessor query is done to find the stored position i that starts the run that contains k to get the entry $N[i] =< \mathsf{SA}[q-1], \mathsf{SA}[q+1] >$. The neighboring values are defined as $\mathsf{SA}[p-1] = \mathsf{SA}[q-1] + k - i$ and $\mathsf{SA}[p+1] = \mathsf{SA}[q+1] + k - i$. The proof for this relies on the fact that contiguous positions in the BWT that are in the same run tend to remain contiguous after applying LF-mapping. This lemma shows that if a `count` query on a pattern returns a BWT interval and a SA value is known for a position in this interval, it is possible to identify the SA values for every position in the interval. Lemma 7 guarantees that a position in the BWT interval is always known. This allows for `locate` queries to be answered with queries to $\mathsf{O}(r)$ space predecessor data structures in $\mathsf{O}(\log\log(n/r))$ time. Gagie et al. also establish that it is possible to increase the size of index in order to achieve optimal time `count` and `locate` queries.

The r-index combines of the SA and BWT samples sampled at run boundaries requiring $\mathsf{O}(r)$ words of space, and supporting `count` queries in $\mathsf{O}(m\log\log(n/r))$ time and `locate` queries in $\mathsf{O}((m + occ)\log\log(n/r))$ time, where m is the length of the query pattern and occ is the number of occurrences of the pattern in the input. In their experiments, r-index outperforms all competing implementations of the FM-index with respect to space-time trade-offs, taking at most the same amount of space and being magnitudes faster in `locate` queries. Compared to indexes based on Lempel-Ziv compression or grammars, r-index uses more space but remains significantly faster for `locate` queries.

3.3 Sampling the Suffix Array: the sr-index

The subsampled r-index, or sr-index, was developed by Cobas et al. [9] to reduce the space requirement of the r-index, which can be large compared to Lempel-Ziv-based indexes when r is large. The main principle of sr-index is to reduce redundancy by removing run

samples that are within a text distance of at most s from another sample. Here, a run sample refers to a value stored for the start of a run in the BWT, whether this is a bit in a bitvector or a SA value. Doing this effectively reduces the space required by r-index from $O(r)$ to $O(min(r, n/s))$. Subsampling is performed by iterating over each run sample t_i for $i = 2, \ldots, r-1$ and checking whether the distance between the run samples t_{i+1} and t_{i-1} is at most s – that is, whether $t_{i+1} - t_{i-1} \leq s$. If the condition is met, then the run sample t_i is removed from the data structures in the r-index that stores information about it. In this way, all run samples are removed so that if the text were segmented into blocks of size $s+1$, there would be no more than two run samples in each block. The *sr*-index additionally introduces a bitvector of size $O(r)$ bits, marking each removed run sample with a 1. The space overhead of this bitvector is negligible compared to the savings gained by removing entries from the r-index data structures, as it is stored in bits rather than full words. Traditionally, the r-index relies on storing SA values at every run boundary to support `locate` queries. As a result, removing some of these run samples requires a modification to the way `locate` queries are performed. In particular, the process of maintaining a toehold SA value for every interval and the process of finding the SA values of all positions in a final BWT interval changes. For example, computing the SA value for the last position of a run is trivial if the run sample for the run was not removed by sub-sampling. However, in the case where the sample was removed, then in order to find the SA for the run boundary the LF-mapping must be applied recursively until a run sample that was not removed is found. The number of steps needed to find a sample is bounded by s, as this is guaranteed by the subsampling process.

The *sr*-index takes $O(m+s)$ time to complete the backward search of a pattern of length m, rather than the expected $O(m \cdot s)$ time. The authors managed to do this by taking advantage of the fact that the only SA values needed are the values of the final interval of the search. To do this, the authors keep track of the last time that the next character in the backward search did not match the character at the position in the BWT of the currently known SA value, i.e., $\mathsf{BWT}[p] \neq P[i]$ as shown in Subsection 3.2. Then, the final SA value can be computed by subtracting the number of LF steps that have occurred since this mismatch from the SA value for this mismatched interval. The remaining step is to find the SA value for the mismatched interval, which can be located in at most s LF steps, resulting in a query time of $O(m+s)$. To `locate` all occurrences, the process involves breaking the final interval into maximal runs and using a single LF-mapping per run to calculate the LF-mapping for every position within the run, as the results are contiguous for positions in the same run. From the subsampling process, all runs will be completed in at most s steps, at which point the SA values can be found by applying ϕ to the LF-mapping results found earlier, given the location of all occurrences. The `locate` query therefore takes $O((m + s \cdot occ) \log\log(n/r))$ time.

As the sampling distance increases, Cobas et al. showed that the space required by *sr*-index reduces but the query times jump to at least three times the r-index query times. The authors also compared the *sr*-index with Lempel-Ziv-based indexes. For most data sets, the Lempel-Ziv indexes required less space than *sr*-index except for a few cases where *sr*-index took less space when using the largest sampling distance. For every data set, *sr*-index was significantly faster in querying than the Lempel-Ziv indexes.

3.4 Sampling the Suffix Array: Access to the Suffix Array via ϕ^{-1}-forest

The r-index achieves significant space savings by leveraging ϕ and ϕ^{-1} to minimize the number of SA values that must be stored. However, in practice, this method of accessing SA is slow because it requires the SA value to be computed for every position between the

desired position and the nearest run boundary position. This led to the development of the ϕ^{-1} forest by Boucher et al. [6] that aims to improve the time required to access the SA by providing random access to the SA.

The ϕ^{-1} forest, which works analogously for ϕ, exploits the iterative nature of ϕ^{-1} to compute random access to SA. The primary idea behind this auxiliary data structure relates to the toehold lemma 7. In particular, $\mathsf{SA}[i+1] - \mathsf{SA}[i]$, or the difference in offsets of two consecutive lexicographical suffixes, does not change as we iteratively set $j = i$, when $\mathsf{SA}[j] = \mathsf{SA}[i] - 1$ and $\mathsf{BWT}[i] = \mathsf{BWT}[i+1]$ for the current j. Consequently, given any text position, it is possible to compute successive values of ϕ^{-1} by iterating over indices and applying the known differences between consecutive SA values, until reaching a run boundary where $\mathsf{BWT}[i] \neq \mathsf{BWT}[i+1]$. Furthermore, when we have a list of SA values at run ends, $\mathcal{E}$, and at run starts, $\mathcal{S}$, ϕ^{-1} can be easily determined for each value in $\mathcal{E}$, since each occurs before some value in $\mathcal{S}$. Using these concepts, ϕ^{-1} can be redefined as follows.

▶ **Definition 9.** *Given a text position $i \in [1..n]$ and a predecessor structure pred, let $s = pred_{\mathcal{E}}(i) \in \mathcal{E}$ and assume that $\phi^{-1}(s)$ is known. Then, $\phi^{-1}(i) = \phi^{-1}(s) + (i - s)$.*

Using this idea, the ϕ^{-1}-graph is constructed with the nodes being the values of $\mathcal{E}$ and edges between nodes occurring when one node is the predecessor of the run following another node (i.e., $\overline{v_i v_j}$ exists when $pred_{\mathcal{E}}(\mathcal{S}[i+1]) = \mathcal{E}[j]$). Each edge is also labeled with a cost, c, and a limit, l, which represent the distances from the positions where a change in predecessor occurs. Each cost is defined by the number of steps the start of the next run takes from its predecessor, or $\mathcal{S}[i+1] - pred_{\mathcal{E}}(i+1)$, and each limit is defined by the distance from the next run, or $succ_{\mathcal{E}}(\mathcal{E}[i]+1) - \mathcal{E}[i]$. An important property of this graph is that every node has a single outgoing edge because the predecessor function is unique, which means that there is no ambiguity in the path to follow when traversing the graph. Boucher et al. show that by following this graph and utilizing the cost and limit values along the edges between nodes on the path, it is possible to determine ϕ^{-1} for any given position in the text. Assume that we are given a SA value, $\mathsf{SA}[s]$, then with the ϕ^{-1} graph, a total of m iterations of ϕ^{-1} can be calculated by traversing a m-length path starting from v_i, where $v_i = \mathcal{E}[i] = pred_{\mathcal{E}}(\mathsf{SA}[s])$. The formula is as follows: $\phi^{-m}(\mathsf{SA}[s]) = \mathcal{S}[i_m + 1] = \mathcal{E}[i_m] + \sum_{k=1}^{m-1} c_{i_k} + c_0$, where $c_0 = \mathsf{SA}[s] - \mathcal{E}[i]$, as long as the requirement is met that $\sum_{k=1}^{j-1} c_{i_k} + c_0 < l_j$ from $j = 2, ..., m-1$, or the sum of costs is less than the limit at every step of the path. This condition determines the length m of the path and also maintains the necessary condition that every node on the path is the predecessor of the SA value found by the previous node.

The ϕ^{-1}-graph is then used to construct the ϕ^{-1}-forest, which allows iterations of ϕ^{-1} to be skipped during the process of finding a missing SA sample in a `locate` query. To do this, long non-overlapping paths from the ϕ^{-1}-graph are extracted and a balanced binary search tree is built on each path such that given $\mathsf{SA}[s]$ it is easy to find the largest index in the path for which $\phi^{-m}(\mathsf{SA}[s])$ can be computed following the limit requirement laid out above. In the balanced binary search tree, each node stores a cost and limit value c and l, respectively. The leaves of the tree are the edges in the path that the tree represents, and therefore their values are easily determined. The internal nodes then represent their cost as the sum of the cost of their two children and their limit as the minimum between the limit of the left child and the limit of the right child minus the cost of the left child, or $min(l_u, l_v - c_u)$. The tree can then be queried with a given SA sample, $\mathsf{SA}[s]$, by first finding the predecessor $\mathcal{E}[i] = pred_{\mathcal{E}}(\mathsf{SA}[s])$ to compute $c_0 = \mathsf{SA}[s] - \mathcal{E}[i]$ and determine the leaf v that corresponds to the edge e_{i_j}, or the outgoing edge of $\mathcal{E}[i]$. The goal of the query is to find the largest k such that $\sum_{x=j}^{m-1} c_{i_x} + c_0$ for all $m = j, ..., k-1$. The algorithm proceeds in two phases: the

first involves traversing the tree from v_i and the second involves descending the tree to a leaf node that represents k. This *fast-forward* query can also be trivially extended to include a constraint, d, such that the result of the query is the largest $k \leq d$. This constraint is important when considering realistic applications in which a specific value within a certain distance from $\mathsf{SA}[s]$ is desired.

The ϕ^{-1}-forest is composed of several of these binary trees, hence its name. To query the forest, with a given SA position, i, the first step is again to determine the predecessor at position, j, and then set $d = i - j$ and $c_0 = \mathsf{SA}[j] - pred_{\mathcal{E}}(\mathsf{SA}[j])$ and $v = e_j$. The trees in the ϕ^{-1} forest are then searched to find one that contains v. If none is found, ϕ^{-1} is computed normally, v is updated to the new node, d is decremented by one, and the trees are searched again for the new v. If v is found in a tree, then the tree is queried as described above and from the result v is updated to the new node, d is subtracted from the number of edges traversed in the query, and the trees are searched again for the new v. This process continues until $d = 0$. In this way, $\mathsf{SA}[i]$ can be computed in $\mathsf{O}((i-j)(\log\log(n/r) + \log r))$ time with $\mathsf{O}(r)$ additional space to the r-index.

Boucher et al. used their ϕ^{-1}-forest to augment the r-index and evaluated its performance against traditional r-index in both the Pizza&Chili repetitive corpus [33] and data sets comprising an increasing number of Chromosome 19 sequences. Their results showed comparable SA access times to the traditional r-index on the Chromosome 19 datasets, and up to a sixfold speedup on the Pizza&Chili datasets. Although this improvement came at the cost of slightly increased memory usage compared to r-index, it marked the first method to improve the calculation of Φ.

4 Acceleration of FM-index via Phrase-Based Techniques

Hong et al. [17] point out that the FM-index can process a `count` query for a pattern P of length m in $O(m)$ time with close to m random accesses to the wavelet trees built on the BWT. In practice, these random accesses tend to be a bottleneck compared to actually using the results of `count` queries since each random access can result in a cache miss due to the poor cache locality of the underlying data structures. The idea was then to reduce the number of random accesses by using a word-based approach where multiple characters are grouped together in the text and in the query pattern. By doing this, the cost of a single backward search step would result in the matching of multiple characters. This reduction in backwards search steps does not resolve the issue of poor cache locality but attempts to mitigate it.

4.1 Grammar Compression for Accelerating FM-Index Backward Search: FIGISS

Deng et al. [10] introduced FM-Indexing Grammars Induced by Suffix Sorting (FIGISS), which aims to accelerate the FM-index by enhancing the backward search step using grammar compression. FIGISS uses a context-free grammar which we briefly define as follows. A context-free grammar is a formal system of production rules that recursively replace terminal and nonterminal symbols with a nonterminal symbol. This is formalized as grammar $G = (V, \Sigma, R, S)$, where V is the set of nonterminal symbols, Σ is a set of terminal symbols, R is the set of production rules that take the form $A \to \alpha$ with $A \in V$ and $\alpha \in (\Sigma \cup V)^*$, and S is the first rule $S \in V$. Their method uses Grammar Compression by Induced Suffix Sorting, known as GCIS [32], which creates a leftmost S-type suffixes (LMS) factorization of an input text T using the linear-time SAIS algorithm [31]. This algorithm works by first

padding the left and right of the input with characters that are lexicographically smaller than all the other characters in the string. Then, it labels each suffix of the input as an S-type suffix or an L-type suffix. This is done by creating a string the same size as the input, iterating through every position i of the input, and adding an S to the string if the suffix starting at that position is lexicographically smaller than the suffix starting at the following position, $i+1$, or by adding an L otherwise. Suffixes that start at $T[0]$ and $T[n+1]$ are also labeled as S-type suffixes. Afterwards, all left-most S-type (LMS) suffixes are defined as every S-type suffix that occurs right after an L-type suffix, where the suffix starting at $T[0]$ is also considered an LMS suffix. The LMS factorization is formed by taking each substring of the created string that starts at one LMS suffix and ends right before the next LMS suffix. Note that the first and last positions of the created string are not included in any substring since they corresponded to the padding characters. Then, each LMS substring is replaced with a nonterminal symbol producing the grammar that is used by FIGGIS.

To perform pattern matching, FIGISS constructs the grammar, $\mathcal{G}_T$, on the input text T, then constructs the grammar, $\mathcal{G}_P$, on the query pattern P, and finally matches P to T by matching their nonterminals. When constructing $\mathcal{G}_P$, the same nonterminal as $\mathcal{G}_T$ is used when the right-hand side of an LMS substring in $T^{(1)}$, the LMS factorization of T, matches the right-hand side of an LMS substring in $P^{(1)}$, the LMS factorization of P. The idea of pattern matching in FIGISS is built on finding a *core*, which is a maximal length substring in $P^{(1)}$ that is contained in $T^{(1)}$ for every occurrence of P in T. Note that the first and last nonterminals $P^{(1)}$ cannot be part of the *core* because the substrings those nonterminals represent do not appear in the same form in $T^{(1)}$. To find this *core*, FIGISS constructs a BWT on $T^{(1)}$, represented as a wavelet tree that is constructed using the colexicographic ranking of the nonterminals. Then, a backward search of $p-2$ steps is done, where p is the number of nonterminals in $P^{(1)}$. The resulting interval represents all occurrences of $P^{(1)}[1..p-1]$ to be found in $T^{(1)}$, which correspondes to all occurrences of $P[1..|P|-1]$ in T. To then find all occurrences including the last nonterminal, $P^{(1)}[p]$, the backward search is started from the intervals defined by any nonterminal of $T^{(1)}$ that contains the LMS substring represented by $P^{(1)}[p]$ as a prefix rather than starting with the entire range of the BWT. In practice, Deng et al. limit the maximum possible size of any LMS substring with a parameter to improve performance and include a generalized suffix tree on the right-hand side of nonterminals to support pattern matching for queries shorter than the maximum length parameter.

Deng et al. evaluated their implementation with different aggregation parameters against RLFM using datasets from the Pizza&Chili repetitive corpus, a data set of 15 Chromosome 19 sequences and a data set of randomly generated strings. FIGISS requires less space than RLFM for every data set, and for every parameter tested, the space decreases as the parameter increases. When answering `count` queries, FIGISS is faster than RLFM for some aggregation parameters when the pattern length is particularly long, but otherwise it is slower than RLFM. Finally, FIGISS requires less total memory during index construction and less final index space but takes significantly more time to construct.

4.2 Prefix-Free Parsing for Enhanced FM-Index Performance: PFP-FM

One of the main limitations of FIGISS was that a single round of LMS factorization led to short phrases that only helped improve the efficiency of `count` queries for particularly long patterns. This is because short phrases do not provide significant speedups from the reduction of backwards search steps compared to the increased overhead required unless the query pattern is long. PFP-FM, an accelerated FM-index, developed by Hong et al. [17]

similarly accelerates the `count` query of the FM-index, but allows for acceleration across a wider range of pattern lengths. This is because PFP-FM uses Prefix-Free Parsing (PFP) which uses user-defined parameters to change the length of the phrases that are created. This allows for longer phrases, which provides a greater reduction in the number of backwards search steps than shorter phrases offer. A detailed description of how PFP works can be found in the preliminaries section 2.7.

The index constructed by PFP-FM consists of the FM-index of T, the FM-index of the parse $\mathcal{P}$, a bitvector $\mathcal{B}$, a hash function h, and a mapping $\mathcal{M}$. $\mathcal{B}$ has the same length as the BWT and stores a 1 for every position for which a rotation of the input starting at that position is prefixed by a trigger string. The hash function h hashes the phrases, and the map $\mathcal{M}$ is used to map the hash of each phrase to the lexicographic rank of the phrase. Hong et al. note that with a sufficiently large hash function, it is possible to store $\mathcal{M}$ in space proportional to the number of phrases while maintaining a negligible probability of a hash collision. This allows the hash function and the map to effectively replace the dictionary $\mathcal{D}$ of the input. Therefore, in order to check if a phrase occurs in the dictionary of the input, the phrase is first hashed and then the map $\mathcal{M}$ is queried with the phrase hash. A null return from the map means that the phrase does not occur in the input.

To perform a `count` query, PFP-FM first parses the pattern P, the same way as the input was parsed using the same rolling hash function. The main difference is that the pattern does not have characters appended to it and is not assumed to be circular, which means that the pattern will not necessarily start or end with a trigger string. Therefore, when P is parsed, there will be up to three resulting components: a prefix that ends at the first trigger string in P, a sequence of complete phrases that are in P, and a suffix that starts at the last trigger string in P. First, if there is a suffix component, it is determined whether that suffix occurs in T by performing a backward search for it in the FM-index of T. If no BWT range is identified, or if a BWT range is identified and the parsed P consists solely of the suffix, then the query process is considered complete. Otherwise, the next step is to determine whether the sequence of phrases in P occurs in T. To do this, the phrases are each hashed with h and the hashes are placed on the map $\mathcal{M}$, which returns NULL if the phrase is not in the input, and their phrase ID if it is. If a NULL value is returned for any phrase, then that phrase and, by extension, P do not occur in T, and the query is done. Otherwise, the sequence of phrases will be encoded in a sequence of phrase IDs. PFP-FM then performs a backward search for this sequence of phrase IDs in the FM-index of $\mathcal{P}$, which means that every backward search step matches a full phrase rather than a single character. The primary acceleration achieved by PFP-FM arises from this mechanism. Continuing with the backward search process, if a BWT range is found and the parsing of P does not have a prefix, the query is complete. Otherwise, the final step is to find the prefix in the parsing of P in T through a backward search in the FM-index of T. In practice, Hong et al. demonstrate that the PFP parameters, w and p, can be adjusted to systematically optimize the performance gains achieved by PFP-FM, as these parameters directly influence the average phrase length.

The authors performed experiments that compared PFP-FM with FIGISS [10], the standard RLFM[26, 22], RLCSA[37], Bowtie[21], and Bowtie2[20] on increasing size SARS-CoV-2 datasets and the human genome GRCH38. In the SARS-CoV-2 datasets, Hong et al. report that PFP-FM requires less cumulative time to complete 1,000 `count` queries on patterns of increasing length than all other competing methods. PFP-FM completes queries at least 451% faster than the second-fastest method, FIGISS. The results were the same on the GRCH38 data set, where PFP-FM was between 3.9 and 7.1 times faster than RLCSA, the second fastest method. This improvement comes at the cost of an increase in index size and memory required during construction.

5 Acceleration via Improving Locality

As shown above, the FM-index and the r-index require several different data structures to support the `count` and `locate` queries. These data structures can become quite large as the size of the input increases, and each must be accessed in succession during query processing. It is difficult to keep all the necessary memory addresses from all different data structures cached, which results in a frequent number of cache misses. This in turn results in slower queries with higher variability. The problem then is to improve the locality of the data in the data structures, such that all the necessary information for a query can be cached together.

5.1 Move data structure

Nishimoto and Tabei [30] introduced the move data structure in 2021. The primary contribution of their paper was to introduce, *OptBWTR*, which is the first string index capable of performing `locate`, `count`, and decompression in $\mathsf{O}(r)$ words of space and optimal time. It also supports extract queries in optimal time with $\mathsf{O}(r + b)$ words of space, where b is some sampling parameter, and prefix search in optimal time with $\mathsf{O}(r' + d)$ words of space, where r' is the number of runs in the RLBWT of the input and d is the size of the set of strings used in the prefix search. This outperforms the r-index, which can attain $\mathsf{O}(r)$ words of space or optimal query times, but not both simultaneously. Additionally, the r-index does not support anything other than `locate` and `count` queries. The index of Nishimoto and Tabei achieves this result through the use of the move data structure, which enables the LF and ϕ^{-1} functions – both essential for `count` and `locate` queries – to be computed in $\mathsf{O}(r)$ words of space and constant time. The previous best space-time tradeoff possible for these functions was in the r-index, which takes $\mathsf{O}(m \log \log(n/r))$ time for `count` queries and $\mathsf{O}((m + occ) \log \log(n/r))$ time for `locate` queries with $\mathsf{O}(r)$ words of space[15].

The primary idea behind the move data structure is that of disjoint interval sequences and `move` queries. A disjoint interval sequence is a way to turn a function's domain and range into a set of non-overlapping input intervals and a set of non-overlapping output intervals. A `move` query takes a position i from the function's domain and the input interval that contains this position, x, and returns an i' and x' such that i' is a new position in the function's domain and x' is the input interval that contains this new position. The way in which i' is determined is by computing $i' = q_x + (i - p_x)$, where p_x and q_x are the start of the x-th input and output interval, respectively. This defines a bijective function that is related in its form to both the LF and the ϕ^{-1} functions. This is significant, as Nishimoto and Tabei demonstrate that dividing the interval sequences to obtain a balanced interval sequence allows `move` queries to be answered in constant time when the move structure is built on such sequences. Since LF and ϕ^{-1} have the same form as the bijective function defined by a disjoint interval sequence and a `move` query, then by building a move structure on the balanced interval sequence of these functions, it is possible to compute them in constant time.

A move data structure is composed of two arrays, pair and index. The pair stores all pairs from the original interval sequence. In the move data structure for the LF function, these pairs are defined as the BWT positions of all run heads and the LF mapping of these positions. The index array stores the index of the interval that contains the second value of the i-th pair. In terms of the LF move structure, this stores the index of the BWT run that contains the LF mapping of the i-th run head. For simplicity, we will now assume that the move structure is a table for which at every position i all the information that would be contained in both arrays is stored. A `move` query on the LF move data structure of some

position i known to be in the input interval x is as follows. First, the LF-mapping of i, let us say i', is determined using the information from the x-th entry of the table by computing $i' = q_x + (i - p_x)$. Then, a linear search is conducted on the table to find the input interval containing i', let us say x', by starting at the input interval that contains the LF-mapping of the start of the x-th run, which is stored by the table. The final step is to return (i', x') for use in future `move` queries that would be necessary in any other backward search. The second step seems like it could be linear in the number of input intervals that need to be searched, but Nishimoto and Tabei established that the number of intervals searched will be constant because the move structure is constructed on a balanced interval sequence.

5.2 RLBWT Tricks

In their paper, RLBWT Tricks, Brown et al. [7] implement the move structure defined by Nishimoto and Tabei, which had yet to be implemented, and refine their approach for practical applications.

First, Brown et al. modified the move table introduced by Nishimoto and Tabei by replacing absolute indices with run-relative start indices. As a result, the table representation requires one fewer column of integers. This change preserves the ability to compute the LF-mapping and ϕ efficiently, but requires an additional sparse bitvector over BWT to mark run starts. This bitvector is used to convert run/offset pairs into absolute text positions at the end of the backward search. Each row in the table is composed of: the character of the run, the length of the run, the index of the run, the LF-mapping of the position at the start of the run, the index of the run that contains the LF-mapping, and the offset into that run it occurs. The main modification that Brown et al. do to the move data structure is to compress the table. Several different methods are presented, implemented, and compared to achieve this. Primarily, the table is partitioned into blocks of fixed size B, allowing each block to be easily loaded into the cache and taking advantage of the inherent locality of the tables for speed-ups. The fixed sizing specifically has the added benefit of positions within the blocks being mapped to easily. Additionally, by using uncompressed bitvectors as a wavelet tree to store the run characters in the block rank, and select queries can be done fast. The authors also augment each block with the position of the first run before and after the current block for each character to allow for quick transitions into other blocks while scanning.

To compress the run-length and offset columns, the authors use directly addressable codes (DACs), which provide a hierarchical encoding of integers [28]. Three different methods are presented to compress the LF-mapping columns. For two of these methods, the property that the LF-mapping values for multiple runs of the same character in a block form a non-decreasing subsequence is exploited. This enables the creation of sublists for each character, which can be represented using partial differences. One method stores these partial differences using DACs and samples the list of differences. The second method represents the partial differences as an uncompressed bitvector, where the number of 0's before each 1 encodes a difference value. For example, to retrieve the fifth value in the difference list, a select query is performed on the bitvector to `locate` the fifth 1. The third method omits partial differences entirely and instead samples the list of differences while applying linear interpolation to recover missing values. Rather than using a single bitvector to represent run head indices, an approach more suitable for large alphabets, a separate uncompressed bitvector is maintained for each character. These per-character bitvectors support faster rank and select queries compared to wavelet trees, at the cost of increased space per character, which is acceptable for data such as DNA that has a small alphabet. Without having implemented the second

half of the Nishimoto and Tabei paper, their implementations rely on sequential scans of the table, which takes $\Omega(r)$ time in the worst case. To combat against large sequential scans, the authors additionally implemented a way to reduce the total number of runs by splitting large runs by some maximum run length during construction.

The implementation developed by Brown et al. is designed to efficiently support `count` queries. Multiple variations of their approach are evaluated, both internally and in comparison to established alternatives such as r-index and RLCSA. The evaluation is carried out on datasets of increasing size, specifically Chromosome 19 and SARS-CoV-2, using randomly selected substrings of varying lengths as query input. Among the tested methods, the variant that employs a bitvector to encode the partial differences list of the LF-mapping values – referred to as *lookup-bv* – demonstrates superior performance relative to all other approaches. For shorter query lengths, the standard *lookup-bv* implementation exhibits the best performance, while for longer queries, a variant that integrates *lookup-bv* with run-splitting techniques proves to be the most effective. In all data sets, the best performing variant of Brown et al. implementation achieves execution speeds comparable to or exceeding those of r-index, although with higher space consumption. Although it is slower than RLCSA, it compensates for it by requiring less space. Overall, the approach proposed by Brown et al. offers a balanced trade-off between time and space efficiency, positioning it between the two competing methods.

5.3 Movi: A fast and cache-efficient full-text pangenome index

Movi, developed by Zakeri et al. [40], is the first full-text pangenome index to utilize Nishimoto and Tabei's move data structure and builds on the work of Brown et al. as well. Movi takes advantage of the locality of reference of the move data structure to reduce the frequency of cache misses and employs a parallelization strategy to hide the latency of any cache misses that do occur. During LF-mapping steps, the move table stores the index of the run that contains the LF-mapping of the current run head. This other run is usually in a different block of the table, and, as such, accessing it results in a cache miss as the new block is retrieved. To hide this, Zakeri et al. use a memory prefetch command and work on multiple `count` queries at the same time. In this way, while one of the `count` queries is waiting for a block to be loaded into the cache, work is still being done on another `count` query, and the latency is not felt. Hiding this latency is crucial as the jump to a run in another block can occur every iteration of the backward search.

During the backward search, we would normally want to use the LF-mapping of the current BWT range indices to find the next BWT range indices. In the move table, we instead use the stored value that indicates the index of the run that contains the LF-mapping of the current run head. However, the BWT index in the current backward search step does not always correspond to a run head, so the run that contains the LF-mapping of the current run head may not contain the LF-mapping of the current BWT index. In order to find the appropriate run, a sequential search through the runs in the table is necessary. Movi provides two different methods for handling the search. The first is a sequential search through the table that checks runs of characters that match the character of the LF-mapping we are looking for to try and find the run that contains this LF-mapping. The second is a constant-time method in which pointers are stored for each character that point to the indices of runs of the character. The locality of reference allows for the sequential search to be practically inexpensive, especially when compared to a cache failure, but the constant-time method is the faster of the two, as the sequential search still has an unpredictable number of memory accesses. However, the constant-time method does have the added cost of being needed to store the pointers.

The authors compare the backward search capabilities of Movi's two methods against the r-index using 10 million simulated 150 bp reads of the seven bacterial species in the Zymo community. The results show that both Movi methods were around 16 times faster than the r-index in completing the query, but required three times as much space in the default method (using sequential search) and 11 times as much space in the constant method (storing pointers).

5.4 Move-r: Optimizing the *r*-index

Move-r is a static text index created by Bertram et al. [4] that incorporates the move data structure described in 5.1 with the r-index described in Subsection 3.2. This allows the Move-r index to be practically faster than other r-index implementations with a small increase in the space requirement. This improvement comes from the ability of the Move-r index to use the move data structure to compute LF and ϕ^{-1} in $O(1)$ through move queries.

One of the primary contributions is the inclusion of an algorithm to balance a disjoint interval sequence on which the move data structure is built. Having a balanced disjoint interval sequence is essential to the proof that allows `move` queries to be answered in constant time. The proposed algorithm works by using two balanced search trees (BST), one for the input disjoint intervals and one for output disjoint intervals. It iterates through every output interval in order and determines whether it is an a-heavy interval by checking if it contains $\geq 2a$ input intervals, where a is some balancing parameter. If it does, then the output interval and related input interval are split, producing an a-balanced interval and another interval that is not necessarily balanced. The algorithm repeats this process until there are only a-balanced intervals in the binary search trees. Each step takes $O(\log k)$ time for BST updates, where k is the number of pairs of intervals in the disjoint interval sequence, and $O(a)$ time to determine if an interval is a-heavy. The authors show that there are at most $\frac{k}{a-1}$ splits, so their algorithm takes $O(k + \frac{k}{a} \log k)$ time and $O(1)$ additional space. The algorithm of Nishimoto and Tabei for constructing the balanced disjoint interval sequence takes $O(k \log k)$ time and $O(k)$ space. The authors also provide a practically optimized `locate` algorithm with an asymptotically worse runtime compared to the `locate` algorithm provided by Nishimoto and Tabei [30]. Their algorithm saves up to $3m$ cache misses, where m is the length of the query pattern, by precomputing and storing the nearest SA value for each run head, which allows constant time lookup during the query rather than accessing multiple different data structures in each iteration of the backward search as is done by *OptBWTR*.

Move-r is compared against the r-index[15], online-rlbwt [2], rcomp [29], lookup-bv [7], and block-rlbwt[11] using Pizza&Chili [33], concatenated samples of chr19, concatenated samples of the SARS-CoV-2 genome, and other repetitive texts. During construction of the index, block-rlbwt tended to take the least amount of time and the least amount of peak memory consumption, but move-r was 0.9-2 times as fast as the other methods, while requiring 0.33-1 times as much peak memory usage. For count queries, block-rlbwt offered the best trade-off between query throughput and index size, and move-r consistently had the highest query throughput but was the third largest index. For locate queries, move-r was 0.8-2.5 times as large and 2-35 times as fast.

6 Future Work

Despite the extensive work that has been done on the FM-index and the field of compressed full-text indexes, there are still future directions to be explored to improve further. One such area is the move data structure by Nishimoto and Tabei [30] and their full-text index.

Current research [7, 40, 4] has led to the implementation of the move data structure, which has been shown to be quicker than the r-index but requires more space. There is still room for improvement than can bring the space requirements closer to the r-index while maintaining the practical query times. Nishimoto and Tabei also provide the theoretical framework for a backward search algorithm and a full-text index based on the move data structure, which has yet to be implemented. Another avenue of work is to dynamically update the move data structure so that new datasets can be added dynamically. Currently, when a new string is being added to the move data structure, it has to be recreated entirely, unlike other indexes that can dynamically change to include new inputs.

Another direction of improvement relates to the ϕ and ϕ^{-1} functions. Traditionally, using these functions requires an iteration for every position between the desired SA value position and the known SA value position. The ϕ^{-1}-forest [6] instead allows some random access to SA using a forest built on the ϕ^{-1} values. The results showed significantly faster query times on small repetitive texts and comparable query times on larger repetitive texts, such as the human genome, at the cost of increased space. Future work in this area could look to improve the random access to ϕ and ϕ^{-1} so that the increased space requirements are reduced and that the performance on SA queries improves over the r-index on larger datasets.

7 Conclusion

In conclusion, the FM-index has profoundly influenced the fields of computational biology, data compression, and text retrieval since its introduction, offering a powerful tool for efficient and compact string indexing. Although the classical FM-index provides an elegant balance between space and time efficiency, the rapid growth in data size and complexity has highlighted its limitations, particularly with highly repetitive and large-scale datasets. This has spurred significant advancements, including the development of run-length encoding, grammar-based compression, and prefix-free parsing techniques, which enhance the FM-index's scalability and performance.

Despite these achievements, the field continues to face challenges that warrant further exploration. Bridging the gap between theoretical innovations and practical implementations remains critical, as is optimizing the balance between memory consumption and query efficiency. Emerging areas such as dynamic FM-indexes, hardware acceleration, and cross-domain applications demonstrate the ongoing evolution of this foundational data structure. As data volumes continue to grow exponentially, the importance of efficient self-indexes, like the FM-index, will only increase. Building upon the innovations reviewed in this paper, researchers have the opportunity to further enhance the capabilities of FM-index variants, addressing contemporary challenges while opening new avenues for applications across diverse domains.

References

1 Alberto Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, pages 85–96. Springer, 1985.

2 Hideo Bannai, Travis Gagie, and Tomohiro I. Refining the r-index. *Theoretical Computer Science*, 812:96–108, April 2020. doi:10.1016/j.tcs.2019.08.005.

3 Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *ACM Transactions on Algorithms (TALG)*, 16(2):1–54, 2020. doi:10.1145/3381417.

4 Nico Bertram, Johannes Fischer, and Lukas Nalbach. Move-r: optimizing the r-index. In *22nd International Symposium on Experimental Algorithms (SEA 2024)*, pages 1–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

5 Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun. Prefix-free parsing for building big BWTs. *Algorithms Molecular Biology*, 14(1):13:1–13:15, 2019. doi:10.1186/S13015-019-0148-5.

6 Christina Boucher, Dominik Köppl, Herman Perera, and Massimiliano Rossi. Accessing the suffix array via ϕ-1-forest. In *International Symposium on String Processing and Information Retrieval*, pages 86–98. Springer, 2022. doi:10.1007/978-3-031-20643-6_7.

7 Nathaniel K Brown, Travis Gagie, and Massimiliano Rossi. Rlbwt tricks. *LIPIcs: Leibniz international proceedings in informatics*, 233:16, 2022.

8 Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, May 1994.

9 Dustin Cobas, Travis Gagie, and Gonzalo Navarro. A fast and small subsampled r-index. *arXiv preprint arXiv:2103.15329*, 2021. arXiv:2103.15329.

10 J. J. Deng, W. K. Hon, D. Köppl, and K. Sadakane. Fm-indexing grammars induced by suffix sorting for long patterns. In *Proceedings of the Data Compression Conference (DCC)*, pages 63–72. IEEE, 2022. doi:10.1109/DCC52660.2022.00014.

11 Diego Díaz-Domínguez, Saska Dönges, Simon J Puglisi, and Leena Salmela. Simple runs-bounded fm-index designs are fast. In *21st International Symposium on Experimental Algorithms (SEA 2023)*, pages 7–1. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.

12 Martin Farach-Colton, Paolo Ferragina, and Shanmugavelayutham Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM (JACM)*, 47(6):987–1011, 2000. doi:10.1145/355541.355547.

13 P Ferragina and G Manzini. Indexing Compressed Text. *Journal of the ACM*, 52:552–581, 2005. doi:10.1145/1082036.1082039.

14 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *the Proceedings of the Foundations of Computer Science (FOCS)*, pages 390–398, 2000. doi:10.1109/SFCS.2000.892127.

15 Travis Gagie, Gonzalo Navarro, and Nicola Prezza. Fully Functional Suffix Trees and Optimal Text Searching in BWT-Runs Bounded Space. *Journal of the ACM*, 67(1):1–54, 2020. doi:10.1145/3375890.

16 Roberto Grossi, Ankur Gupta, Jeffrey Scott Vitter, et al. High-order entropy-compressed text indexes. In *SODA*, volume 3, pages 841–850, 2003.

17 Aaron Hong, Marco Oliva, Dominik Köppl, Hideo Bannai, Christina Boucher, and Travis Gagie. Pfp-fm: an accelerated fm-index. *Algorithms for Molecular Biology*, 19(1):15, 2024. doi:10.1186/S13015-024-00260-8.

18 Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi. Permuted longest-common-prefix array. In *Proceedings of the Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192, 2009. doi:10.1007/978-3-642-02441-2_17.

19 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006. doi:10.1145/1217856.1217858.

20 B Langmead and S L Salzberg. Fast gapped-read alignment with {B}owtie 2. *Nature Methods*, 9(4):357–359, 2012.

21 Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25–R25, 2009. Publisher: BioMed Central. doi:10.1186/gb-2009-10-3-r25.

22 Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. In *Annual Symposium on Combinatorial Pattern Matching*, pages 45–56. Springer, 2005. doi:10.1007/11496656_5.

23 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.

24 Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976. doi:10.1145/321941.321946.

25 J Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 408–424. SIAM, 2017. doi:10.1137/1.9781611974782.26.

26 V Mäkinen and G Navarro. Run-length FM-index. *the Proceedings of the DIMACS Workshop: The Burrows-Wheeler Transform: Ten Years Later*, pages 17–19, 2004.

27 Gonzalo Navarro. Indexing text using the Ziv–Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004. doi:10.1016/S1570-8667(03)00066-2.

28 Gonzalo Navarro. *Compact data structures: A practical approach*. Cambridge University Press, 2016.

29 Takaaki Nishimoto, Shunsuke Kanda, and Yasuo Tabei. An optimal-time rlbwt construction in bwt-runs bounded space. *arXiv preprint arXiv:2202.07885*, 2022. arXiv:2202.07885.

30 Takaaki Nishimoto and Yasuo Tabei. Optimal-time queries on bwt-runs compressed indexes. *arXiv preprint arXiv:2006.05104*, 2020.

31 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE transactions on computers*, 60(10):1471–1484, 2010. doi:10.1109/TC.2010.188.

32 Daniel Saad Nogueira Nunes, Felipe Louza, Simon Gog, Mauricio Ayala-Rincón, and Gonzalo Navarro. A grammar compression algorithm based on induced suffix sorting. In *2018 Data Compression Conference*, pages 42–51. IEEE, 2018. doi:10.1109/DCC.2018.00012.

33 Pizza & Chili repetitive corpus. Available at http://pizzachili.dcc.uchile.cl/repcorpus.html. Accessed 16 April 2020.

34 Alberto Policriti and Nicola Prezza. Lz77 computation based on the run-length encoded bwt. *Algorithmica*, 80(7):1986–2011, 2018. doi:10.1007/S00453-017-0327-Z.

35 Nicola Prezza. Compressed computation for text indexing, 2017.

36 Kunihiko Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In *International symposium on algorithms and computation*, pages 410–421. Springer, 2000. doi:10.1007/3-540-40996-3_35.

37 J Sirén. Compressed suffix arrays for massive data. In *International Symposium on String Processing and Information Retrieval*, pages 63–74, 2009.

38 Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995. doi:10.1007/BF01206331.

39 Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the Annual Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE, 1973. doi:10.1109/SWAT.1973.13.

40 Mohsen Zakeri, Nathaniel K Brown, Omar Y Ahmed, Travis Gagie, and Ben Langmead. Movi: a fast and cache-efficient full-text pangenome index. *iScience*, 27(12), 2024.